Tractor Hacking

Jin Huang and Kyle Kesler

Senior Project Report

Electrical Engineering Department

California Polytechnic State University

San Luis Obispo

2021

Table of Contents

Lists of Tables and Figures

Tables:

Figures:

Abstract

Tractor hacking arose from tractor owners' desire for full control over the tractors they purchase. Tractor manufacturers set strict security on the electrical components of their tractors, making it difficult for tractor owners to diagnose and monitor their tractors on their own. The UltraBlue Tractor Hacking project is a solution to this problem, allowing tractor owners to perform diagnostics and monitor the equipment they purchased without having to go through dealerships. From talking to farmers, we found that they valued a diagnostic tool that allows for clearing and resetting Diagnostic Trouble Codes (DTCs), viewing diagnostic data pertaining to a specific fault that was detected by an electronic control unit (ECU) in the tractor, and obtaining a repair plan from the diagnostics data. This project is an attempt to provide tractor owners and technicians the hardware and software necessary to diagnose, monitor, and repair their equipment without the need for manufacturer technicians and repair services. Our product is a device that utilizes the SAE J1939 standard to implement diagnostic and monitoring functionalities on tractors that conform to the SAE J1939 standard. Both the hardware and software have gone through rigorous testing and validation to ensure that we are following the standards set by SAE J1939. After that was completed, we tested our device on John Deere tractors to test the effectiveness of our implementation of a diagnostic tool. However, the results of these tests led us to conclude that not all John Deere tractors conform to the SAE J1939 standard.

Introduction

The increasing complexity of automotive technology in the past twenty years has led to increased difficulty in diagnosing faults and performing repairs on all vehicles. In the case of tractors and other farming vehicles, companies like John Deere have taken almost complete control of their products even after their sale to customers. Primarily, John Deere holds a large amount of power over the repair and maintenance of all their sold vehicles. As of 2021, this remains true and tractor owners find it increasingly expensive and difficult to have their vehicles repaired [1]. The process often involves contacting a John Deere technician, waiting for them to attempt a diagnosis using JD Service Advisor, then, if needed, coming for a physical inspection, ordering the brand new "required" replacement part, making the repair, and then calibrating/reprogramming the new part, ultimately costing thousands for a sometimes simple repair.

The tractor hacking project is dedicated to the research of tractor computer systems for the purposes of repair. The previous iteration of this project was able to read CAN-bus messages and extract a portion of the desired data. It was unable to read DTCs, clear/reset DTCs, and request specific data from an ECU, all functionalities required to provide a viable diagnostic tool .

This shows a clear need for our project to be able to read DTCs, clear/reset DTCs, and read live data. In order to implement these functionalities, we make the assumption that all John Deere tractors conform to the SAE J1939 standard, a standard implemented in heavy machinery equipment manufactured after 1994. We hope to push the project further by creating a diagnostic tool that follows the SAE J1939 standard and can perform similar functions to JD Service Advisor. With this tool, we can determine if John Deere tractors, as well as tractors from other manufacturers conform to the standard. If they do, it would be a first step in giving the right to repair back to the tractor owners and diminishing the monopoly that tractor manufacturers have over its products.

The following sections of Chapter I describe key terms and technologies involved with a tractor's OBD system. Chapter II covers analysis of the marketing requirements and the engineering design specifications determined based on the marketing requirements. A functional decomposition using block diagrams is provided in Chapter III along with descriptions of the functional relationships between components. Chapter IV provides the completed project schedule, while Chapter V describes the design and construction process of the finished product. Finally, Chapter VI covers the verification of the design and results from tests on different tractors.

Chapter I: Background

Controller Area Network (CAN) Bus

The Controller Area Network (CAN) bus is a communication protocol that facilitates the communication of OBD data between Electronic Control Units (ECUs) within a vehicle [2]. The CAN-bus links various ECUs within the vehicle to facilitate efficient and fast data transfer. The CAN-bus message consists of a message identifier (29-bits for J1939) corresponding to the ECU the signal came from and the message data which is the information provided by that particular ECU or what data is being requested from that ECU [3]. Figure 1 depicts the CAN bus network in a standard vehicle, which is relatively similar to the CAN bus network of a tractor. The CAN bus communication protocol was first implemented to minimize the wiring required to connect nearly 250 sensors found in modern vehicles [2]. Sensors and actuators are tied to ECUs that preside over a subnetwork of the vehicle, which are then tied to CAN bus lines connected to main ECUs. The main ECUs are what can be accessed through the vehicle diagnostic port [4].

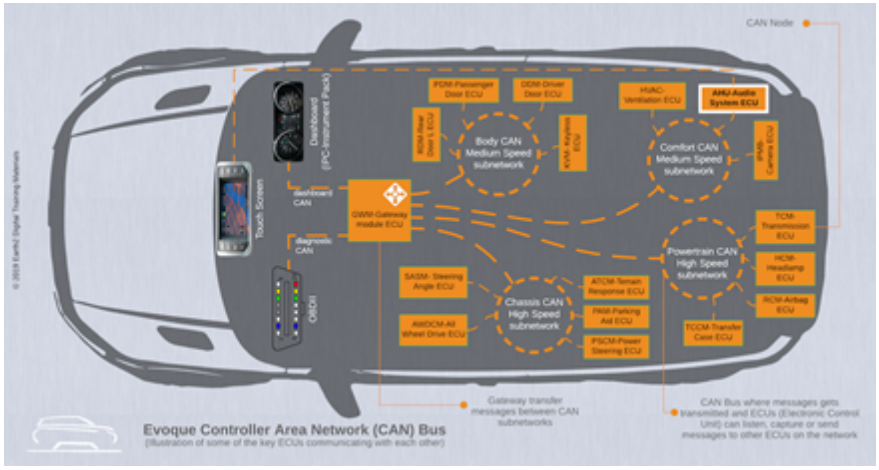


Figure 1: CAN bus Network in a Standard Vehicle

CAN Message

A CAN message consists of many different parts. The most important parts are the arbitration field (identifier), data length code (DLC), data, and the cyclic redundancy check (CRC). The J1939 standard uses a CAN extended data frame that consists of a 29-bit identifier instead of the 11-bit identifier used in a typical CAN data frame [5]. Figure 2 below shows the breakdown of a CAN message.

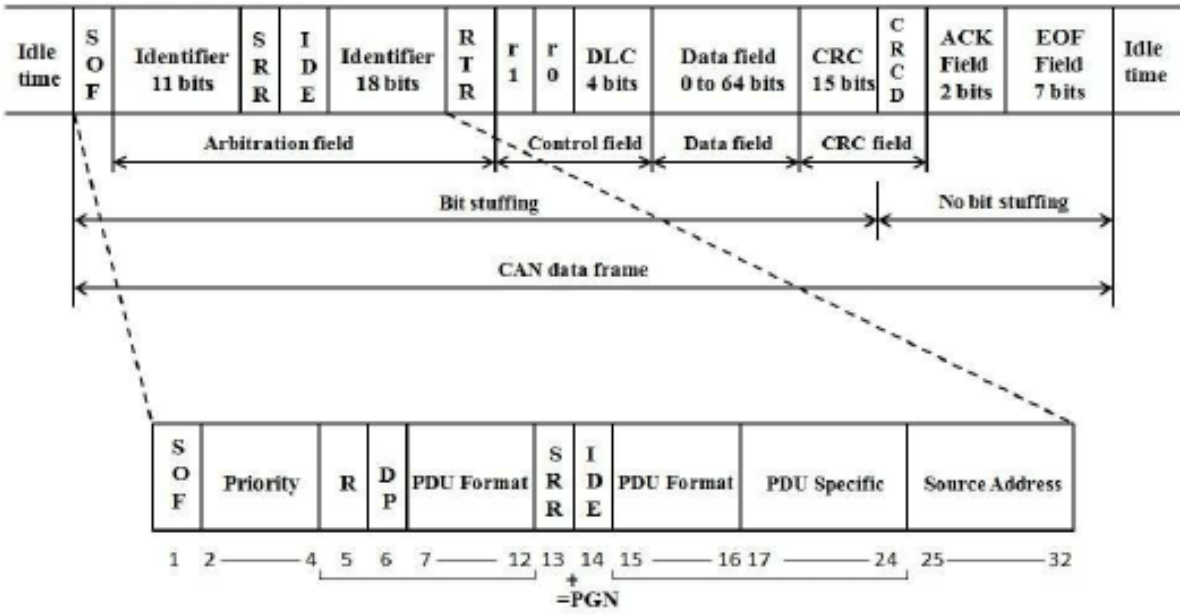


Figure 2: CAN Extended Data Frame

Identifier

The 29-bit identifier is broken down in the bottom of Figure 2. The priority determines the importance of the CAN message, the PDU format determines if the message has a destination address or is a broadcast message, and the PDU specific contains the destination address or group extension depending on the PDU format [6]. Together the PDU format and specific combine to address a specific PGN corresponding to the targeted ECU(s) [7]. The source address is the unique address of the ECU sending the CAN message.

Data Frame

A CAN data frame includes the identifier, the data field, the CRC field, and the acknowledgement field. The identifier is explained in the section above. The data field contains 0-8 bytes of data that can be extracted and processed for information. The CRC field is used for error checking to verify data integrity. The acknowledgement field is used to indicate whether or not an ECU has acknowledged and received data correctly [5].

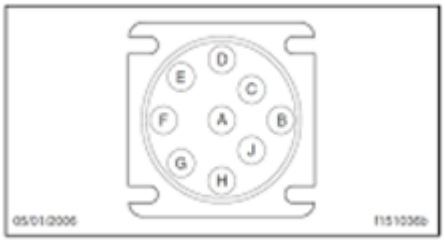
Diagnostic Trouble Codes (DTCs)

Diagnostic trouble codes are individual codes related to a specific error in a vehicle. The codes are used by the vehicle's OBD system to notify the user of an error, such as a yellow or red indicator light on a car's dashboard [4][8]. DTCs can be used to determine where a fault occurred and what data is needed to narrow down the fault to a specific cause [3].

On-Board Diagnostic (OBD)

Beginning in 1962, every vehicle in the U.S. was designed with an OBD system [9]. OBD allows the vehicle to self-diagnose and provide status reports to a port that the user can connect to. Figure 3 displays the J1939 9 Pin Female to open end cable, while Figure 4 displays the pinout of the J1939 connector.

Figure 3: J1939 9 Pin Female to Open End Cable



Diagnostic Connector	
Pin	Function
A	Battery (–)
B	Battery (+)
C	J1939 CAN High (+)
D	J1939 CAN Low (–)
E	Reserved
F	J1708/U1587 (+)
G	J1708/U1587 (–)
H	Diagnostic CAN High (+)
J	Diagnostic CAN Low (–)

Figure 4: J1939 9 Pin Female Connector Pinout

In the J1939 pinout defined in Figure 4, pins C and D correspond to J1939 CAN-H and CAN-L which are the desired communication pins. Pins H and J correspond to the diagnostic CAN-H and CAN-L which do carry CAN bus messages, but not the diagnostics data that is required for the purpose of this project [8][9][10]. The J1939 connector also has a ground and 12V supply battery pin supplied by the engine battery [10].

Using the J1939 female to open end cable, the user can read, clear, and reset DTCs to gain valuable information that can be used to determine the location and cause of a fault [3][11].

Chapter II: Requirements

The requirements and specifications for this project were generated through background research on similar existing products (i.e. automobile OBD2 bluetooth adapters), interviews conducted on target customers, and certain constraints that pertain to this particular product. More information about the interviews can be found in the appendices under the Customer Interviews section.

TABLE I

Ultrablue Tractor Hacking Requirements and Specifications

Marketing Requirements	**Engineering Specifications**	**Justification**
1	Device enclosure is rated IP68	The encasing of the device should provide adequate protection in order to prevent damage from the environment that the device will be operating in.
2	Device dimensions should not exceed 2”x4”x3/4”	The device should be medium sized so that it is mobile and not easily lost.
2, 5	Utilizes Bluetooth Low Energy (BLE) for data transmission	Transmission of data through Bluetooth allows for the bypass of physical storage on the device, minimizing the risk of mechanical failure while also reducing the size of the device. BLE allows for low energy data transmission, minimizing power consumption.
3, 4	Follows J1939 standard	J1939 is a standard used in heavy-duty vehicles. Utilizing this standard will allow for general usage of this device across different tractor brands and models.
3, 4	Operates from 12V and has to be able to tolerate a ripple of 50-100mV	The device is to be plugged into a tractor’s OBD-II port, which is powered by 12V and has a ripple of 50-100mV when the alternator is running.

3, 4, 6	Performs diagnostics tests, clears and resets codes, provides solutions	The device should be able to perform multiple functions that are useful in aiding in the maintenance, repair, and upgrade of tractors.
2, 5	Connects to a new mobile application (designed to have an interface similar to JD Service Advisor) via Bluetooth LE	A mobile application similar to JD Service Advisor allows for an interface that tractor technicians are familiar with. Everything that the device transmits from the tractor via Bluetooth will be accessible from the mobile application.
Marketing Requirements 1. Robust and durable 2. Mobile 3. Connects through the OBD-II port 4. Universal Usage 5. Bluetooth Enabled 6. Multi-functional		

TABLE II

Ultrablue Tractor Hacking Deliverables

Delivery Date	Deliverable Description
2/5/21	Design Review
3/22/21	EE 461 demo
3/20/21	EE 461 report
10/26/20	ABET Sr. Project Analysis
6/4/21	EE 462 demo
6/8/21	EE 462 Report
6/11/21	Submit Senior Project

Chapter III: Functional Decomposition

The following chapter contains the functional decomposition of the project in the form of a level 0 and level 1 block diagram, illustrated in Figures 5 and 6, respectively. Together, they illustrate the overall functionality of the device and the inputs and outputs of the system.

Level 0 Block Diagram

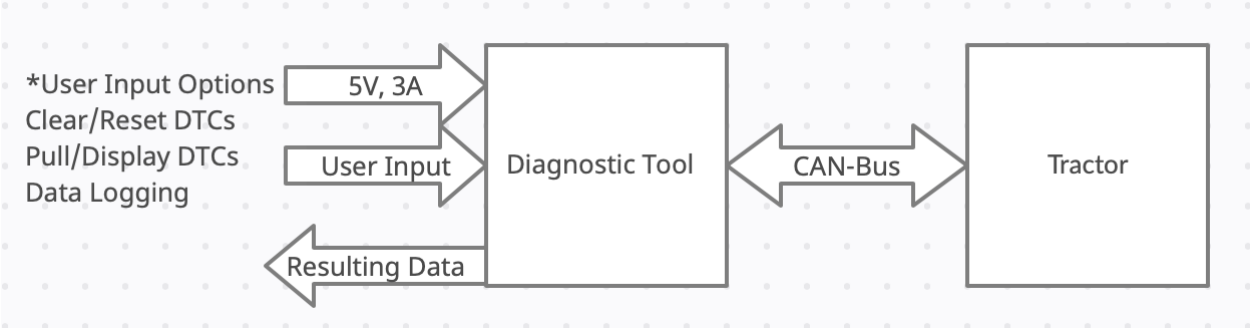


Figure 5: Level 0 Block Diagram

TABLE III: Level 0 Diagnostic Tool

Module	Diagnostic Tool
Inputs	Clear/Reset DTCs
	Pull and Display DTCs
	Data Logging
	CAN Bus Message
	5V 3A DC Power
Outputs	CAN Bus Message
	Resulting Data

Functionality	The Diagnostic Tool receives user input to determine which mode of operation the user desires. Then using the determined mode of operation, it communicates with the tractor via CAN bus messages to obtain the necessary data or fulfill the desired function. The resulting data is then displayed on the Diagnostic Tool's terminal. Power is supplied through a DC power cable supplying 5 volts 3 amps.

TABLE IV: Level 0 Tractor

Module	Tractor
Inputs	CAN Bus Message
Outputs	CAN Bus Message
Functionality	The tractor communicates with the Diagnostic Tool via CAN bus messages. The tractor either provides the data requested or it completes a command sent by the diagnostic tool.

Level 1 Block Diagram

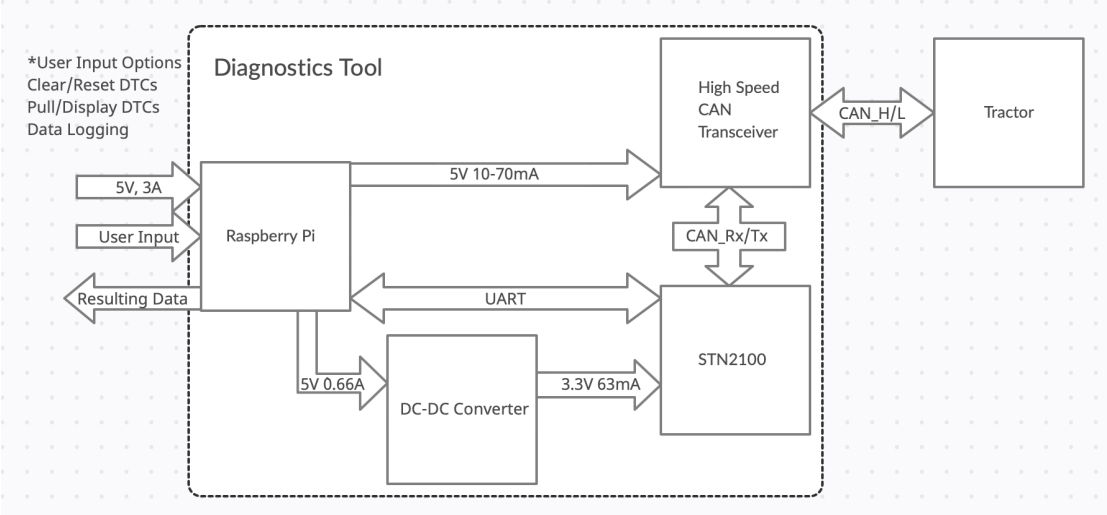


Figure 6: Level 1 Block Diagram

TABLE V: Level 1 Raspberry Pi 4B

Module	Raspberry Pi 4B
Inputs	User Input
	5V 3A DC Power
	UART
Outputs	UART
	5V 70mA DC Power
	5V 41.6mA DC Power
Functionality	The Raspberry Pi 4B takes user input to determine which mode of operation the user desires. Using the determined mode of operation, it communicates with the STN2100 via UART to obtain the necessary data or fulfill the desired function. The resulting data or status is then displayed on the Raspberry Pi's terminal. Power is supplied to the Raspberry Pi through a DC power cable supplying 5 volts 3 amps. The Raspberry Pi supplies 5 volts 41.6 milliamps to the DC-DC converter and 5 volts 70 milliamps to the High Speed CAN Transceiver.

TABLE VI: Level 1 DC-DC Converter

Module	DC-DC Converter
Inputs	5V 41.6mA DC Power
Outputs	3.3V 63mA DC Power
Functionality	The DC-to-DC converter steps down the 5V power supplied by the Raspberry Pi to 3.3V for the STN2100. The STN2100 requires 3.3V at 63mA so the Raspberry Pi supplies 41.6mA at 5V.

TABLE VII: Level 1 STN2100

Module	STN2100
Inputs	UART
	CAN_RX
	3.3V 63mA DC Power
Outputs	CAN_TX
	UART
Functionality	The Raspberry Pi 4B communicates with the STN2100 via UART using ST/AT commands as well as J1939 OBD messages to obtain the necessary data or fulfill the desired function. These commands and messages are then sent to the High Speed CAN Transceiver via the CAN_TX output to be transmitted to the tractor so that the appropriate data can be requested. The High Speed CAN Transceiver then sends that data back to the STN2100 via the CAN_RX input to be processed and sent back to the Raspberry Pi 4B via UART to be displayed on the terminal.

TABLE VIII: Level 1 High Speed CAN Transceiver

Module	High Speed CAN Transceiver MCP2562
Inputs	CAN_RX
	CAN Bus Message
	5V 70mA DC Power
Outputs	CAN_TX
	CAN Bus Message
Functionality	The High Speed CAN Transceiver takes in CAN messages from the STN2100 via a serial CAN_RX and translates the message for transmission on the CAN-Bus (CAN_H/CAN/L). Responses from the tractor are read over the CAN-Bus and sent in serial format back to the STN2100 via CAN_TX.

TABLE IX: Level 1 Tractor

Module	Tractor
Inputs	CAN Bus Message
Outputs	CAN Bus Message
Functionality	The Tractor communicates with the High Speed CAN Transceiver via the CAN-Bus line. The tractor either provides the data requested or it completes a command sent by the diagnostic tool.

Chapter IV: Project Planning

Our project followed an agile development methodology called scrum, a software development framework that utilizes sprints. Sprints are a short, predetermined time period where our team completes a set amount of work. Each sprint is organized in a way that work is targeted towards specific tasks pertaining to a feature, or features if the workload permits. At the end of the sprint, we return to the product owners for evaluation of the features and developments completed in that sprint. After a team reflection on the past sprint, we add items to the next sprint using feedback from product owners and technical knowledge to determine the best course of action and features for the development of the product. Figures 7 through 9 depict our progress throughout the project.

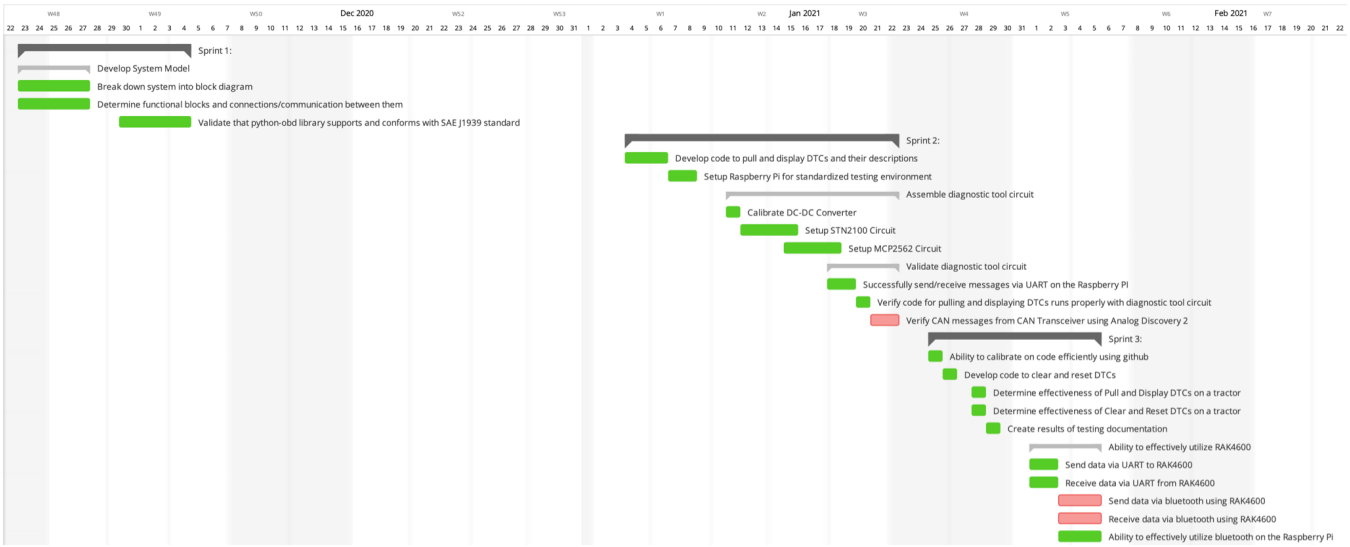


Figure 7: Sprints 1 through 3

Sprints 1 through 3 were heavily focused on background research that allowed us to get started on development and construction in sprint 3. This included familiarizing ourselves with datasheets, looking into the specifications and requirements for each part and how they come to function together.

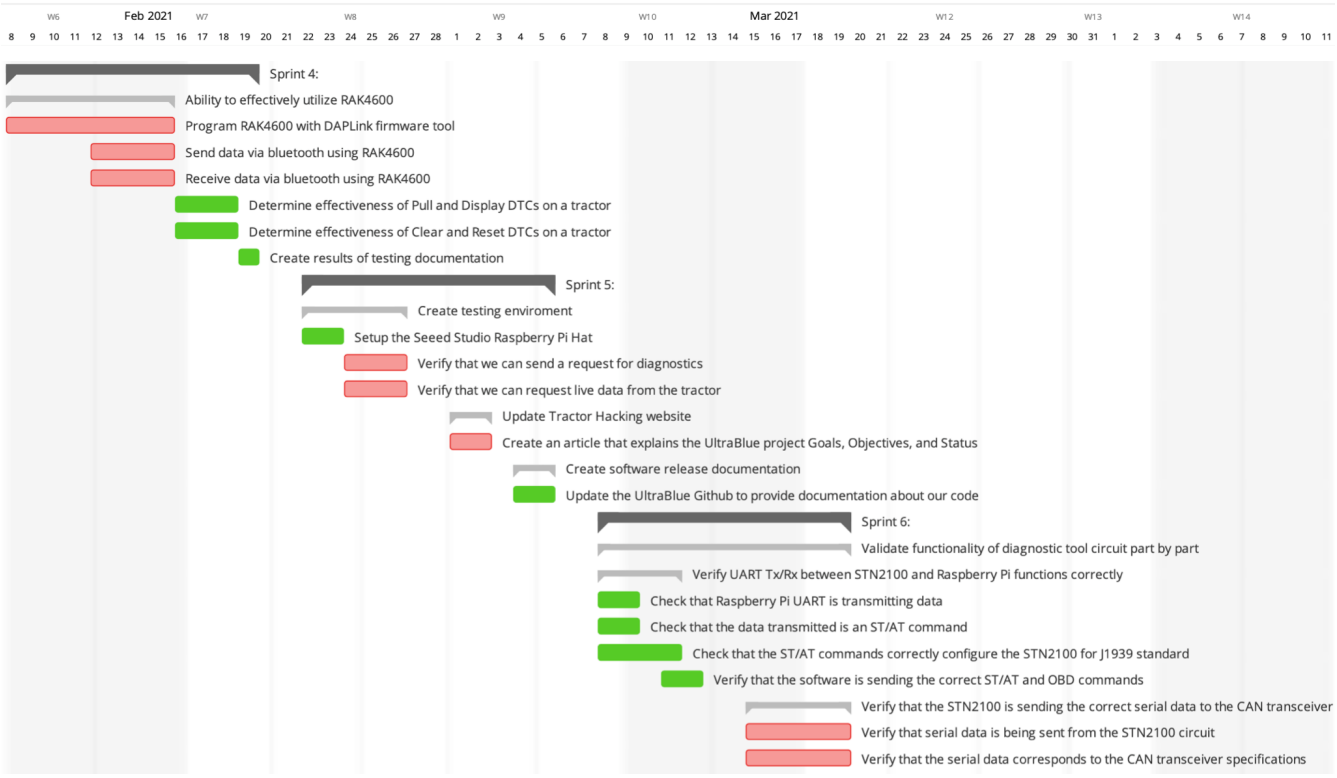


Figure 8: Sprints 4 through 6

Sprint 4 was spent attempting to implement bluetooth as well as testing the core functionality that we developed thus far. Sprint 5 was focused on documentation as well as creating a testing environment.Sprint 6 was dedicated to validating the functionality of the STN2100 circuit and Raspberry Pi 4B.

Figure 9: Sprints 7 through 10

Sprint 7 was dedicated to integrating a new CAN transceiver chip into the circuit. Sprint 8 was spent revalidating the circuit with this new part to ensure it is properly functioning. Sprint 9 was used to run multiple tests and document our results. Given sprint 9's results, we were able to modify our setup to attempt different methods for testing during sprint 10.

Chapter V: Development and Construction

Design Overview

The design of this project aims to meet the defined marketing requirements and engineering specifications in order to satisfy the target customer. We were unable to meet all the requirements and specifications which will be discussed in a later section. Figure 10 below shows the final product and our configuration of the Raspberry Pi 4 with the CAN transceiver and OBD to UART interpreter.

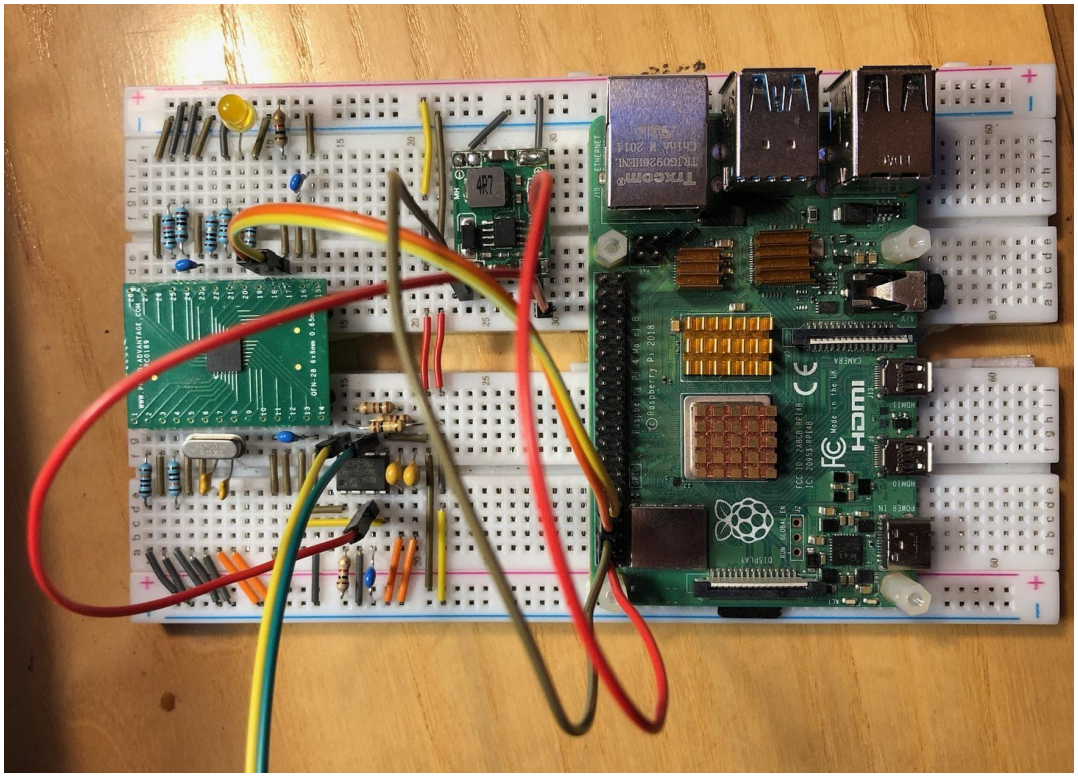


Figure 10: Tractor Hacking Final Product

Hardware

Raspberry Pi 4B

The Raspberry Pi 4B is a Single Board Computer that serves as the environment for the development and application of software, and acts as the main interface between the other hardware and outside components such as bluetooth transmission to a smartphone app. The Raspberry Pi is powered by a 5V 3A DC power cable and provides the MCP2562 High Speed CAN Transceiver with 5V 10-70mA and the DC-DC Voltage Converter with 5V 41.6mA (which in turn powers the STN2100 with 3.3V 63mA). When a desired function is selected, the Raspberry Pi communicates with the STN2100 via UART using ST/AT commands to select desired settings as well as OBD CAN messages to communicate with the tractor and request the desired data or action.

STN2100 Multiprotocol OBD to UART Interpreter

The STN2100 is a Multiprotocol OBD to UART interpreter capable of implementing the SAE J1939 OBD-II standard and transmitting/receiving data via a built-in CAN-bus. The STN2100 requires 3.3V and an operating current of 63 mA.

To set up the circuit for the STN2100 follow the wiring diagram in Figure 11 below. The only I/O pins used are the CAN_RX/GP8, CAN_TX/GP9, UART TX, UART RX, Power_CTRL/GP19, VDD, and GND, so all other control pins should be connected according to the minimum connections wiring diagram in Figure 12. The I/O pins should be connected as follows:

CAN_RX/GP8 → RXD (MCP2562)

CAN_TX/GP9 → TXD (MCP2562)

GND → GND

UART RX → UART TXD (Input)

UART TX → UART RXD (Output)

VDD → 5V

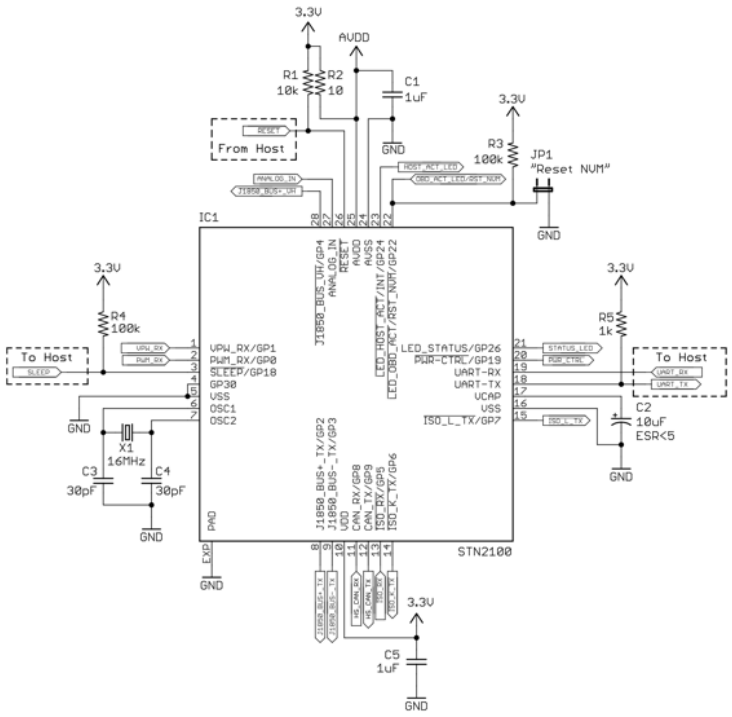


Figure 11: STN2100 Standard Connections Diagram

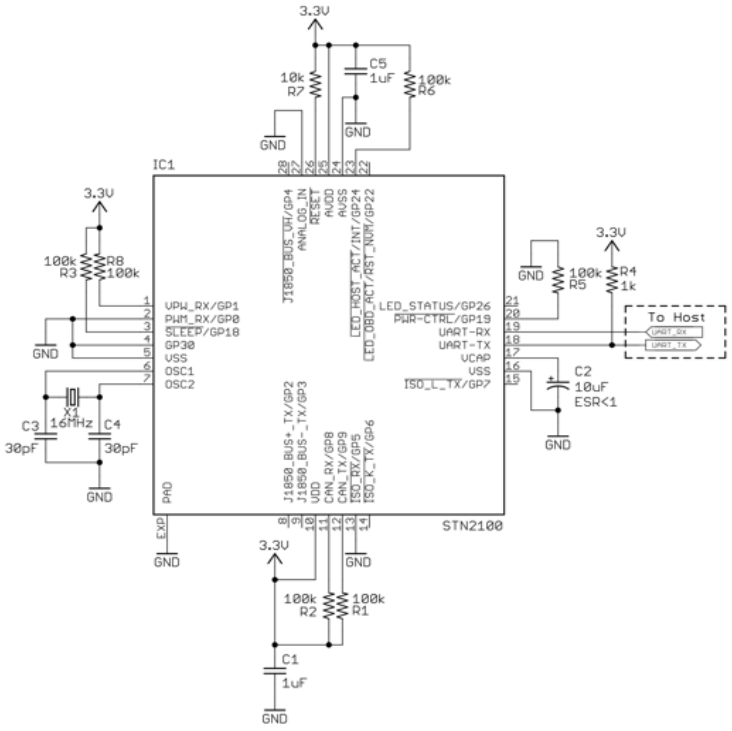


Figure 12: STN2100 Minimum Connections Diagram

MCP2562 High-Speed CAN Transceiver

To enable CAN-bus communication using the STN2100 we opted to use the MCP2562, a High Speed CAN Flexible Data Rate Transceiver. This transceiver acts as an interface between the CAN protocol controller, the STN2100, and the physical bus on the tractor. It converts the digital signals generated by the CAN controller into a differential output signal used for transmission over the bus cabling. The MCP2562 requires 5V and 10 - 70mA. In this use case a 120Ω resistor is placed between CANH and CANL. The rest of the pinout is as follows:

VDD → 5V

VSS → GND

VIO → 3.3V

TXD → CAN_TX/GP9

RXD → CAN_RX/GP8

CANH and CANL are connected to the corresponding CAN-bus lines of the tractor. Use Figures 13 and 14 for reference.

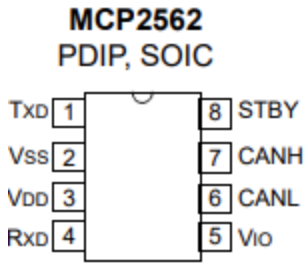


Figure 13: MCP2562 Pinout

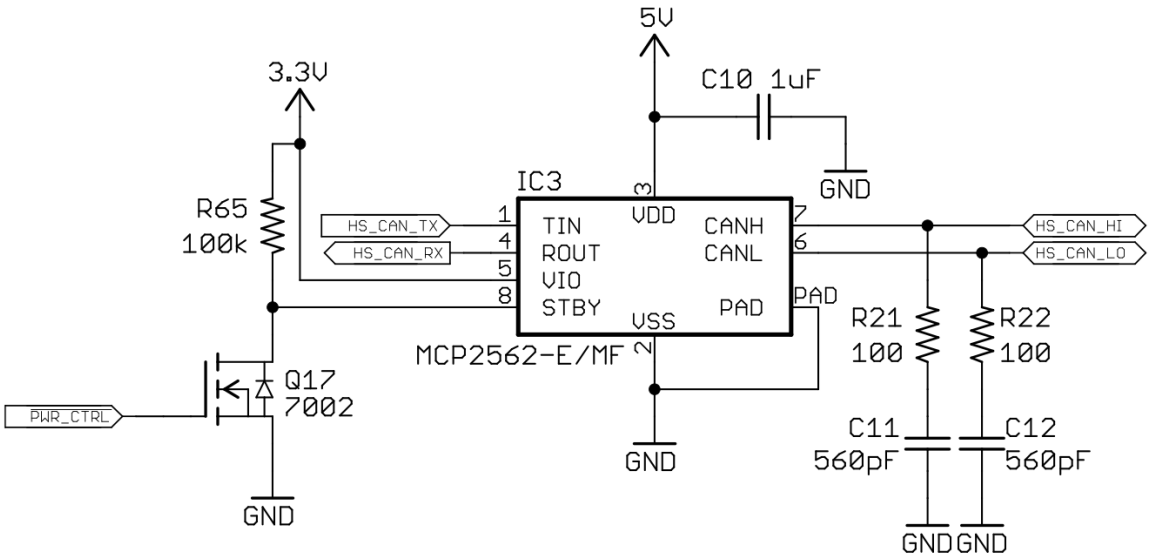


Figure 14: MCP2562 Connection Diagram

Software

The software for this project utilizes the python-obd library to handle data from the tractor. It can live stream sensor data, perform diagnostics, and is made for use with the Raspberry Pi. The library is designed to work with standard ELM327 adapters, which the STN2100 supports. For each of the functions, the parameter "portstr" in the connection variable needs to be changed accordingly. The parameter "protocol" in the connection variable is set to be "A" since "A" corresponds to SAE J1939 in the library.

Pull and Display DTCs

The software for this function pulls and displays DTCs reported by the tractor. The code will print the response of the get DTC command, which should list the DTC code and a description (if the library has information about it).

Clear/Reset DTCs

The software for this function clears/resets DTCs reported by the tractor. The code will print the response of the clear DTC command, which should print nothing if the operation was successful. Clearing/resetting DTCs only clears/resets the DTC until the tractor is turned off. If the DTC is fixed, the DTC will not be reported, but if it isn't, then the DTC will be reported again.

Live Data

The software for this function pulls and displays live data for certain parameters. Currently, the code includes engine temperature, speed (in km/h), RPM, fuel level, and oil temperature, but more can be added. The code will endlessly pull whatever parameters it is given every 3 seconds. The time between requests can also be adjusted.

Challenges and Setbacks

Bluetooth Low Energy (BLE) was part of the original design, however, we found that the selected bluetooth module was not feasible for the functionality required for the diagnostic tool. Instead, we decided to focus on the core functionality of our product to ensure that we had an operational product before adding features that weren't a necessity, such as bluetooth. With that being said, we also cut out the case design, as we felt it was impractical to design a case without a first working product to encase. Similarly, without proper function of the hardware and data to be sent to a mobile application the implementation of a smartphone app was less of a priority.

Initially we assumed that the Seeed Studio 2-Channel CAN-bus Shield would allow us to create a complete test environment by using the CAN-bus shield to mimic the tractor portion of the system. As we were attempting to do so, we found that the software in the library we were using required a connection to an actual vehicle in order for the necessary portion of the code to execute. During the implementation process, we also discovered that the CAN-bus shield was using CAN FD, a protocol typically used in modern high performance vehicles, which is unsupported by the logic analyzer we were using.

As described in Test 2 of the Chapter VI Test Results, during a test we shorted pins on a New Holland tractor's diagnostic port and blew four diagnostic port fuses. We were quick to have the tractor repaired and reach out to all parties involved. As a result, we halted testing and shifted our focus to the full verification of our system. Before any future tests, we planned more in depth procedures for testing and more thorough test plans.

Another setback was we initially selected an MCP2561 FD as our CAN transceiver, but because it has a SPLIT pin for common mode stabilization instead of a VIO digital I/O supply pin. We instead chose the MCP2562 that has a VIO pin so that the I/O pin logic level could be adjusted from 5V down to the STN2100s logic level of 3.3V.

Another big challenge we faced was COVID. COVID restrictions have posed a great challenge for development as well as testing. We didn't have access to the laboratory equipment we normally would be able to utilize. Testing our product on campus was also much more difficult to arrange due to the university's policies on accessing the campus and its resources.

Chapter VI: Verification and Test Results

Verification

We broke down our system and identified key portions that could be tested individually for proper functionality.

We began with verifying that UART communication on the STN2100 circuit functions correctly. We did this by checking that the Raspberry Pi 4B's UART was transmitting data, checking that the data transmitted was an AT command, and checking that the AT command was the correct command for the STN2100. The next step was to verify that the STN2100 IC was sending the correct data to the MCP2561 CAN transceiver. This included verifying that data is being sent from the STN2100 circuit, verifying that the data meets the CAN transceiver specifications, and verifying that the CAN transceiver transmits the correct data packet on the CAN-bus lines. Lastly, we wanted to verify that the software was sending the correct commands by looking through the python-obd library code and checking that it conforms to the standard.

When verifying that the Raspberry Pi 4B UART was transmitting data, we were able to send and receive messages within the Raspberry Pi's UART, as well as from the Raspberry Pi's UART to the STN2100. We discovered that the default baud rate was 9600 kilobytes per second, which was far less than the 115,200 kilobytes per second baud rate we were originally trying to transmit data at. When verifying that the STN2100 was sending the correct data to the CAN transceiver, we found that we were not able to see any meaningful data on the Analog Discovery 2 Module's protocol analyzer. We only saw errors that matched when data was received in the code. We realized that the CAN transceiver we were using was not fit for this application, since it was shifting to 3.3V rather than 5V. We switched to the MCP2562 CAN transceiver, which does shift to 5V and saw much better results. The last verification step for UART communication was verifying that the software was sending the correct commands. We sifted through various SAE J1939 standards to determine exactly what should be sent, and found the exact command that should be sent. Figure 15 below shows the output of the software after successful communication (sending AT commands and receiving responses).

```
[obd.obd] ====================== python-OBD (v0.7.1) ======================
[obd.obd] Explicit port defined
[obd.elm327] Initializing ELM327: PORT=/dev/ttyS0 BAUD=9600 PROTOCOL=A
[obd.elm327] write: b'ATZ\r'
[obd.elm327] wait: 1 seconds
[obd.elm327] read: b'ATZ\r\r\rELM327 v1.3a\r\r>'
[obd.elm327] write: b'ATE0\r'
[obd.elm327] read: b'ATE0\rOK\r\r>'
[obd.elm327] write: b'ATH1\r'
[obd.elm327] read: b'OK\r\r>'
[obd.elm327] write: b'ATL0\r'
[obd.elm327] read: b'OK\r\r>'
[obd.elm327] write: b'AT RV\r'
[obd.elm327] read: b'0.0V\r\r>'
[obd.elm327] OBD2 socket disconnected
[obd.obd] Cannot load commands: No connection to car
[obd.obd] ================================================================
[obd.obd] Sending command: b'03': Get DTCs
[obd.elm327] write: b'03\r'
[obd.elm327] read: b'SEARCHING...\rUNABLE TO CONNECT\r\r>'
```

Figure 15: Console Output of Software for UART Communication Verification

The next step was to integrate the new MCP2562 CAN transceiver chip into the circuit. We analyzed the output of the system in order to confirm that requests conform to the SAE J1939 diagnostic standard. We did this by repeating the same steps we took for the MCP2561 CAN transceiver chip. We were able to do so, shown in Figure 16 below.

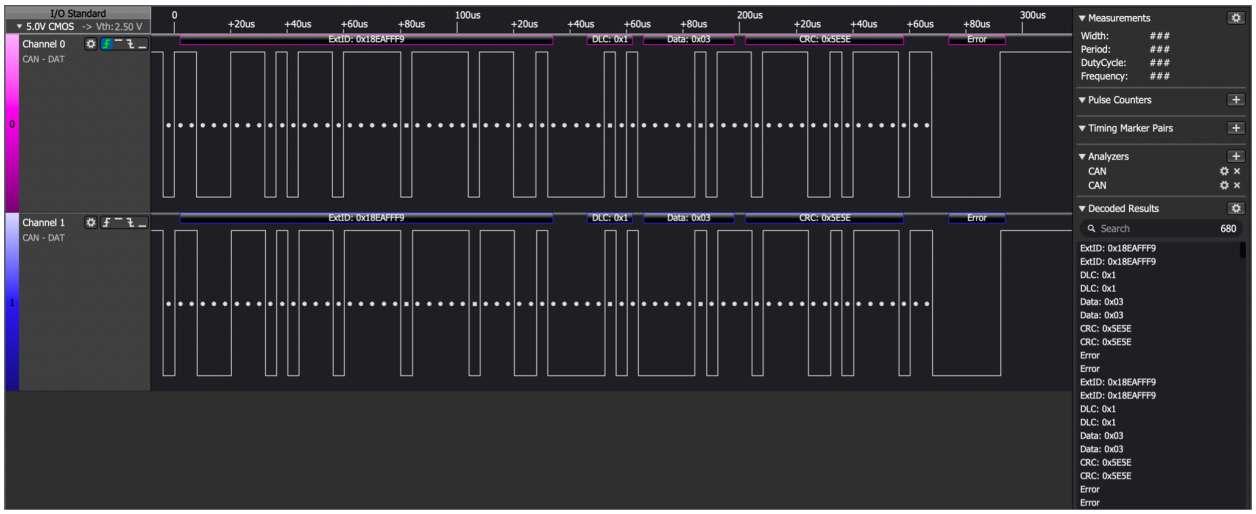


Figure 16: Decoded CAN Transceiver Output Verification Using A Logic Analyzer

Test Results

Test 1

To test our final product, we performed a series of tests on tractors at Cal Poly. Our first test consisted of running a full systems test. The goal of the first test was to determine the effectiveness of using our hardware and software to pull and display DTCs from a tractor. Being able to verify that our setup is able to successfully do so was an important milestone in our project, as it is a major first step in communicating with the tractor. The test was conducted on a New Holland tractor at the Cal Poly Dairy Farm on January 30, 2021.

We were unable to determine the effectiveness of using our code and circuit to pull and display DTCs. We encountered many problems as we went, which ultimately led to the inability to properly test our setup.

The first problem we encountered was that the circuit was not being powered properly. We overlooked the ability of the USB port of the laptop to supply sufficient amperage to power the circuit resulting in the status LED turning on despite being unable to function properly. This problem cost a lot of time since it was not discovered until well into the testing session. Once we realized this, we opted to power the circuit using the tractor's 12V battery instead, using the SeeedStudio CAN-BUS shield to step down the voltage from 12V to 5V.

However, we were able to confirm that the tractor port did not follow the SAE J1939 pinout standard. This was discovered when powering the SeeedStudio CAN-BUS shield through the 12V pin on our J1939 connector. Although 12V was mapped to pin B in the SAE J1939 standard, it was actually mapped to pin

H on this tractor's J1939 port. The CAN_H and CAN_L bus lines of the connector also did not follow the pinout shown in Figure 4, so we had to manually determine which pin corresponded to which allocation. A 120Ω impedance is standard for a CAN bus, so when one is connected to another they form a parallel network of two 120Ω impedances in parallel, resulting in 60Ω. For the New Holland tractor we tested on, we found that pins B and J were CAN_H and CAN_L, though we were unable to determine the polarity.

Unfortunately, at this point the laptop we were using to run the code ran out of battery, so we were unable to fully test our code and circuit. This concluded our first test session.

Test 2

The goal of the second test was to again determine the effectiveness of using our code and circuit to pull, display, clear, and reset DTCs from a tractor, as well as to pull and display live data.. The test was conducted on a New Holland tractor at the Cal Poly Dairy Farm, and on a John Deere tractor at the Cal Poly Student Experimental Farm on February 19, 2021.

We were unable to determine the effectiveness of using our code and circuit to pull, display, clear, and reset DTCs, nor were we able to pull and display live data. We first performed tests on the New Holland tractor's diagnostic port. After finding the power pin and wiring it to the Seeed Studio board, we turned the tractor on to check that the Seeed Studio board was outputting the correct voltage. It was at this point that we noticed that the dashboard display did not light up, and the tractor did not seem to be supplying 12V. We turned the tractor off and disconnected what we found to be the power pin from the Seeed Studio board. We then turned the tractor back on and tested the rest of the pins and found that no pin was supplying 12V. Upon learning this, we decided to conduct the same tests on the John Deere tractor. After testing on the John Deere tractor's diagnostic port, we could not find the power pin on any of the pins. We were able to determine the CAN_H and CAN_L bus lines by looking for resistances and voltages indicative of a CAN network though. We returned to the New Holland tractor to perform the same tests, but yielded the same results.

In hindsight, while we were trying to determine the power pin for the New Holland tractor, the wires on the connector may have shorted, which blew four fuses on the tractor. We promptly had a technician service the tractor and paid for the replacement of the fuses.

We determined that there is an issue with the J1939 port on the John Deere tractor, since we were unable to locate the power pin despite testing all pins. However, we were able to determine that the CAN_H and CAN_L bus lines on the John Deere tractor did not follow the SAE J1939 standard.

It was through this test that we discovered that the SeeedStudio CAN-Bus shield was unnecessary when testing the core functionality of this product. We also realized that running a full systems test in one test session was a little too ambitious, so we scaled back and decided to validate parts of the circuit individually before continuing with testing.

Test 3

The goal of the third test was to determine the pinout of the diagnostic port of the tractor. We believed that this was the best first test to conduct in our series of incremental tests. The test was conducted on a tractor at the Cal Poly Farm Shop. We were able to determine the pinout of the diagnostic port on the John Deere tractor, and found that it followed the pinout shown in Figure 2. Pin B read 12V, which is the battery of the tractor, pin C read 2.5V, which is CAN_H, and pin D read 2.3V, which is CAN_L.

Test 4

The goal of the fourth test was to determine the protocol that the data from the tractor conforms to. The test was conducted on the same tractor as the previous test at the Cal Poly Farm Shop. We used the J1939 9 Pin Female to Open End cable and a Kingst LA1010 Logic Analyzer to capture data from the tractor's diagnostic port. With that data capture, shown in Figure 17 below, we were able to decode the data and break it down according to the CAN message section. After decoding the data, we learned that the tractor's data did adhere to the SAE J1939 standard, and that it used a 29-bit identifier.

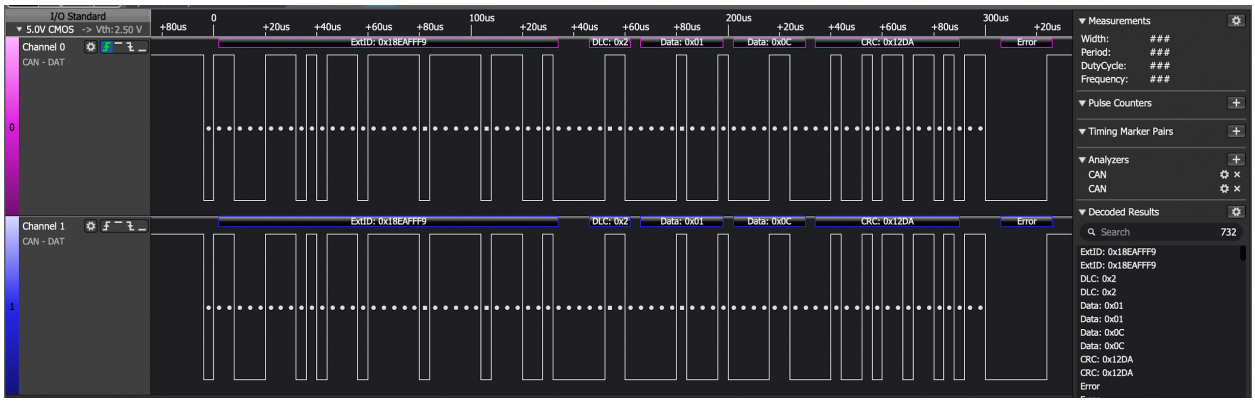


Figure 17: Logic Analyzer Data Decode

Test 5

The goal of the fifth test was to determine the effectiveness of our hardware and software to pull and display DTCs. The test was conducted on the same tractor as the previous test at the Cal Poly Farm Shop. This tractor had no DTCs, which allowed us to test our software to see if it would correctly report no DTCs. Figure 18 shows the output of our code, correctly reporting that there were no DTCs on the tractor.

```
[obd.obd] Sending command: b'03': Get DTCs
[obd.elm327] write: b'03\r'
[obd.elm327] read: b'\r>'
[obd.obd] No valid OBD Messages returned
No DTCs
```

Figure 18: Output for Pull and Display DTC Output

Test 6

The goal of the sixth and final test was to determine the effectiveness of our hardware and software to pull and display DTCs on a tractor that has live DTCs, get live data, and clear/reset DTC. The test was done on a different John Deere tractor at the Cal Poly Farm Shop. We repeated all the procedures starting from the third test. We began by determining the pinout of the diagnostic port of the tractor. After verifying that it matched that of Figure 4, we moved on to determining the protocol that the data of the tractor conforms to. The output of the logic analyzer is shown in Figure 19. The output of the logic analyzer in Figure 19 does not resemble accurate CAN data packets, which is most likely due to this model of tractor using CAN flexible data rate.

Figure 19: Bad Logic Analyzer Read of Tractor OBD Messages

Despite the inability to read clean CAN data packets using the logic analyzer, we moved on to test the functionality of our software. We were again able to test that the tractor was able to correctly report no DTCs. This time, we were able to easily disconnect a sensor on the tractor to force a DTC. Unfortunately, through many different attempts to pull and display the DTC, we were unable to. We attempted to pull the DTC using the python-obd library, as well as through a couple different AT/ST commands, but none of these methods reported a DTC.

We were also able to read diagnostic data from the tractor's Engine Control Unit One (ECU1), as shown below in Figure 20. The software continually monitors for messages from ECU1 until the buffer fills. This demonstrates our ability to read SAE J1939 packets from the CAN bus.

```
[obd.elm327] write: b'ATR 0\r'
[obd.elm327] read: b'OK\r\r>'
[obd.elm327] write: b'ATMP F004\r'
[obd.elm327] read: b'F0 FF 7D 00 00 FF FF FF \rF0 FF 7D 00 00 FF FF FF \rF0 FF 7
D 00 00 FF FF FF \rF0 FF 7D 00 00 FF FF FF \rF0 FF 7D 00 00 FF FF FF \rF0 FF 7D
00 00 FF FF FF \rF0 FF 7D 00 00 FF FF FF \rF0 FF 7D 00 00 FF FF FF \rF0 FF 7D 00
 00 FF FF FF \rF0 FF 7D 00 00 FF FF FF \rF0 FF 7D 00 00 FF FF FF \rF0 FF 7D 00 0
0 FF FF FF \rF0 FF 7D 00 00 FF FF FF \rF0 FF 7D 00 00 FF FF FF \rF0 FF 7D 00 00
```

Figure 20: Engine Control Unit One Data

Chapter VII: Conclusion

Through extensive research and testing, we were able to design and build an embedded system that implements the SAE J1939 OBD standard to provide diagnostic capabilities such as reading DTCs, clearing and resetting DTCs, and reading live diagnostic data. Our test results did not prove the capabilities we aimed to provide in our diagnostic tool, which leads us to the conclusion that not all John Deere tractors conform to the SAE J1939 standard, disproving our initial assumption.

Despite the lack of diagnostic results in tests 4-6 which implement our final design, we are confident that our design implements the SAE J1939 diagnostic standard and can communicate with a tractor that also follows the standard to provide the diagnostic functions previously outlined. This confidence is due to the rigorous validation process our implementation underwent in which we verified the correct functionality of each component, as well as of each component communication intersection. We also verified the final output via logic analyzer to ensure that the CAN data packet sent to the tractor conforms to what is expected for that diagnostic request in regards to the standard.

Though the results of our testing were disappointing, we discovered valuable knowledge about John Deere's tractors. Our conclusion that John Deere tractors do not utilize the SAE J1939 diagnostic standard demonstrates John Deere's attempt at preserving their monopoly over repairing their equipment. While this iteration of this project was unable to successfully prove our implementation of a tool that is able to communicate with John Deere tractors, it provides a foundation for others to expand upon. Future iterations of this project could continue to expand on the features of this project and potentially test on tractors that do conform to the SAE J1939 standard, or focus solely on John Deere tractors and cracking the communication protocol that they utilize.

Bibliography

[1] K. Wiens, “We Can't Let John Deere Destroy the Very Idea of Ownership,” Wired, 21-Apr-2015. [Online]. Available: https://www.wired.com/2015/04/dmca-ownership-john-deere/.

[2] A. S. Siddiqui, Y. Gui, J. Plusquellic and F. Saqib, "Secure communication over CANBus," *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Boston, MA, 2017, pp. 1264-1267, doi: 10.1109/MWSCAS.2017.8053160.

[3] T. U. Kang, H. M. Song, S. Jeong and H. K. Kim, "Automated Reverse Engineering and Attack for CAN Using OBD-II," *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*, Chicago, IL, USA, 2018, pp. 1-7, doi: 10.1109/VTCFall.2018.8690781.

[4] Santini, A., 2011. *OBD-II: Functions, Monitors and Diagnostic Techniques*. Australia: Delmar/Cengage Learning.

[5] “J1939 Introduction,” *Kvaser*, 12-May-2021. [Online]. Available: https://www.kvaser.com/about-can/higher-layer-protocols/j1939-introduction/. [Accessed: 10-Mar-2021].

[6] H. Yun, S. Lee and O. Kwon, "Vehicle-generated data exchange protocol for Remote OBD inspection and maintenance," *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, Seogwipo, 2011, pp. 81-84.

[7] S. Corrigan, “Introduction to the Controller Area Network (CAN),” *Texas Instruments*, Aug-2002. [Online]. Available: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1622102496201&ref_url=https%253A%252F%25. [Accessed: 10-Mar-2021].

[8] Eric Walter; Richard Walter, "OBD-II Diagnostic Messages and Test Modes," in *Data Acquisition from Light-Duty Vehicles Using OBD and CAN*, SAE, 2018, pp.39-51.

[9] Eric Walter; Richard Walter, "On-Board Diagnostics (OBD) Background and Standards," in Data Acquisition from Light-Duty Vehicles Using OBD and CAN, SAE, 2018, pp.25-37.

[10] Dgtech.com. 2014. *DG Technologies Product Pinouts and Industry Connectors Reference Guide*. [online] Available at: https://www.dgtech.com/wpcontent/uploads/2016/04/Pinouts_ICR.pdf

[11] Eric Walter; Richard Walter, "Data Acquisition from Light-Duty Vehicles Using OBD and CAN," in Data Acquisition from Light-Duty Vehicles Using OBD and CAN, SAE, 2018, pp. I-xv.

[12] *Surface Vehicle Recommended Practice - On Board Diagnostics Implementation Guide*, J1939/3_201511, 2015

[13] P. R. Burje, K. J. Karande and A. B. Jagadale, "Embedded On-Board Diagnostics system using CAN protocol," *2014 International Conference on Communication and Signal Processing*, Melmaruvathur, 2014, pp. 734-737, doi: 10.1109/ICCSP.2014.6949940.

[14] P. R. Sawant and Y. B. Mane, "Design and Development of On-Board Diagnostic (OBD) Device for Cars," *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, Pune, India, 2018, pp. 1-4, doi: 10.1109/ICCUBEA.2018.8697833.

Appendices

Senior Project Analysis

Summary of Functional Requirements

The scope of this project focuses on providing tractor technicians with the minimum set of tools necessary for a farmer to run basic diagnostics on their tractor. This project gives the user the ability to pull and display DTCs, clear and reset DTCs, and get live data. The product will be able to implement said functionality by utilizing the SAE J1939 standard to communicate with the tractor.

Primary Constraints

The main challenge relating to this project is the inability to access OBD (onboard diagnostics) data, given that power is currently only held by the manufacturer. Without the right to repair, farmers find accessing OBD data and diagnosing their vehicles quite difficult. This creates spinoff problems in each of the functionality requirements this project aims to achieve. One such constraint on our current implementation is that it is only compatible with tractors that follow the SAE J1939 standard. Another limitation is that the diagnostic tool does not support CAN flexible data rate, so the data rate of the CAN bus must be selected beforehand.

Economics

This product enables John Deere tractor owners and technicians to access their equipment to diagnose problems and get live data; all things that are currently done by a John Deere certified technician that costs hundreds to thousands of dollars on a per repair basis. Tractor owners would be able to do all the repairs themselves or through third-party technicians, significantly reducing repair costs for tractor owners and creating job opportunities for third party technicians.

All of the components used in this design are common, easily accessible parts. The project requires a J1939 to open end cable, a Raspberry Pi 4B, a CAN transceiver, and a multiprotocol OBD to UART module. All of these components are manufactured on a large scale and do not require any adjustments from their original design. Additionally, the assembly of this product is very simple and can be done with minimal manual labor. As with any electronic device, there is electronic waste at the end of the product's lifetime. With this project the typical life expectancy exceeds 10 years, given its fixed state upon the product creation. The product is connected to the tractor via a J1939 cable and remains as a permanent attachment in a weather resistant encasing. Since the majority of the product is the open-source code used, the product could remain in use through continuous software updates to improve the functionality of the product. There is also no need to use any hardware upgrades on a regular basis, which results in less silicon extraction and manufacturing.

Costs and benefits accrue at the end of the project lifecycle. Since the majority of the product revolves around open-source software. The final implementation of all combined physical components will not occur until the software has been tested, verified, and is ready for programming of the final device.

The inputs required are user driven to determine what the user's current needs are from the attached hardware. This project is designed for tractor owners seeking a faster and less expensive method for vehicle diagnosis and monitoring. These tractor owners would be responsible for covering the costs in real world applications, but in the scope of this project, costs are covered by Pinnacle Systems Group, iFixit, and Baer Repairs, who all have invested interests in the power of third-party repairs. All equipment costs are outlined in the cost estimates section of this report.

This project is for the Right-to-Repair movement in their fight to combat large companies like John Deere that have a monopoly over their products and repairs [1]. As stated previously, products will emerge once the final product is capable of providing the capabilities that other third-party alternatives have created and is comparable to John Deere's Service Advisor.

If Manufactured on a Commercial Basis

Estimated Total Serviceable Market for agricultural and non-agricultural equipment (2019-2020) is $420.9 Billion USD. The total US serviceable market for agricultural and non agricultural equipment (2019-2020) is $71.2 Billion. However, because our hardware module is connected to the diagnostic port, it can be placed on systems going back to the 1990s making the total aggregate market far greater.
The global market for diagnostic repair software is roughly $185 Billion. The US market for diagnostic software is conservatively $36.3 Billion in 2019-2020 alone. We believe we can service a minimum of 20% of this market. Total manufacturing costs would be around $185.00 which is made up almost entirely of component costs. If the device were to be manufactured in bulk, component prices would drop as well. The estimated purchase price for each device is $250.00 which includes a markup of 35.14% to account for shipping, taxes, and a profit margin. Even at $250.00, this device is considerably less expensive than comparable devices such as JD Advisor - which starts at $700.00 - and cuts down on repair costs to a fraction of what they are going through John Deere. Technicians typically charge $100-$200 per hour, including travel to and from the job site. Renting a typical replacement tractor generally costs $1000 a week. This means that overall maintenance and repair costs for a small farmer are around $3,000/month, with medium operations facing overall breakdown costs of roughly $50,000/month. Given the easily accessible mobile application, the cost for the user to operate this device could be anywhere from 5 minutes to a few hours depending on the use and diagnostics report that comes back.

Environmental

This project uses electricity from the tractor. For the component connected to the tractor, it will be powered through the tractor's 12V battery which is in turn powered by the tractor's engine. The device is indirectly powered by the tractor, so it won't add any additional waste beyond what the tractor itself is already creating. This being harmful CO2 emissions that negatively impact air quality, ozone layer, and health of animals and plants in the direct vicinity.

Manufacturability

The manufacturing process could be automated once the circuit is designed on a printed circuit board, which would speed up production, minimize labor costs, and minimize the environmental impacts relating to IC production.

Sustainability

Since the device requires no component upgrades, it does not pose any serious sustainability concerns. Once the product is produced, all parts should remain functional and to par with required computing power. Software can be updated at any time for the purpose of this product, and hardware should not require any replacement. If a new standard for OBD or SAE J1939 were to be implemented, it is likely that either a new product would need to be developed (if CAN is no longer used) or a software update would be required to be used on any future vehicles that do not follow the specified standard.

The 12V tractor battery source that powers the device is not sustainable and is powered by a diesel engine which has serious drawbacks for the environment in the form of harmful emissions. As vehicles transition to electric power, tractors might transition to electric power as well which would leave the disposal of a lithium battery having the most serious impact on the environment. Again, both the gas burned by the engine and the battery are already in use when they are powering the tractor.

As far as upgrading the device, the hardware components are encased in a weather resistant box, which can be opened to replace components inside. If the water sealant rubber on the encasing is damaged that will need to be replaced.

Individual components do have their own impact. The production of electronics, plastics, rubber, and other materials used in this project have yet to be produced in a sustainable manner. Plastic is likely the most sustainable of the materials requiring the least amount of energy for production while also being the most recyclable, followed by rubber and then electronics. Most of the materials used in this project are recyclable, allowing for the repurposing of things like electronics and plastics into other electronics and plastics.

Ethical

The UltraBlue project raises important ethical implications, especially about the use of unauthorized products and repairs. Currently John Deere has in place protections so that only registered technicians can service their products. Following the right-to-repair movement, this project aims to give that power back to the owners and third-party repair services so that one company doesn’t have a monopoly over everything relating to their products and can’t force unnecessary upgrades and replacements on their users.

From a utilitarian point of view, this product will make tractor owners happy but make tractor manufacturers unhappy. There are far more tractor owners than tractor manufacturers, so ethically, from a utilitarian point of view, the creation of this product promotes the greater good. Of course, this issue is much more complicated than is explained. Some tractor owners do not have the technical knowledge to

use such a product and would prefer to go through a dealership, even if it may cost more in time and money. Tractor manufacturers also employ many people who would rather not see the decline of their company which could cost them their jobs.

As far as the use of the device, following the IEEE code of ethics falls on the user. Given access to others farming equipment, a user equipped with this device could potentially perform malicious changes to other's equipment. Additionally, users are responsible for understanding the information presented to them, and understanding the privacy and safety of the users, others, and the environment. In order to protect the users and others from harm (IEEE Code of Ethics #9), our product will provide accurate and unbiased data.

Health and Safety

The delivered device is intended to be used at any time during the tractor's lifespan. In some cases, the tractor will be in use while the device is being used as well. Since the user interacts remotely through a smartphone, it is important to ensure that the user is not in the vicinity of the tractor path or operating the app while on the tractor. Use while the tractor is idle or working autonomously is acceptable. Additionally, this device is intended for use by a trained professional or an individual with sufficient knowledge of tractor systems and repair. It is the user's responsibility to ensure that all diagnoses are handled by a trained/experienced professional. Misuse of this product could include intentionally clearing DTCs that require attention without doing anything about them. This could cause the tractor to become non-operational and malfunction, causing harm to both the tractor and the tractor operator.

Social and Political

This project has the power to significantly impact not only agriculture, but big business as a whole. The direct stakeholders are tractor owners, tractor technicians, Baer Repairs, and John Deere. Tractor owners and tractor technicians benefit the most from this project, as they would be able to diagnose their own equipment, which would save thousands of dollars in time and money. Tractor technicians would also benefit from the creation of more third-party repair jobs, since more than just John Deere employees could make diagnoses. John Deere would be very negatively affected by this project as they will suffer a dramatic decrease in diagnostic service income and employees. The indirect stakeholders are big business companies such as Apple that hold monopolies over the repair and upgrade of their products. Other big businesses fear the right-to-repair movement started by tractor owners because from a legal standpoint it could drastically affect their business as well. If the right-to-repair movement gains more significant legal ground other companies who monopolize repairs will face the same backlash from their users.

Using the Raspberry Pi 4 to power this project could cause a problem for the Raspberry Pi Foundation. If this project were to gain a lot of traction in the agricultural world, tractor manufacturers could attempt to shut it down to withhold the right to repair from their customers. The Raspberry Pi Foundation could become a target, as their products are what is powering the products that are taking from the tractor manufacturer's business. People who are attempting to hack other products may also look to use the Raspberry Pi too, and if those people are successful, it could cause the Raspberry Pi Foundation additional trouble.

Development

The SAE standards were exceptionally helpful in learning more about SAE J1939 standards and the information that can be found from a vehicle's ECU. Understanding the variety of sensors and measurements that come together to form DTCs, which manifest to the user in the form of warning lights or indicators, led to a fuller picture of what kind of diagnostics data can be obtained and how to narrow down possible faults through correlated data extraction. We learned more about the framework behind CAN messages, and what data can be extracted to get diagnostic data. We also learned more about accessing data from an ECU to get live data from the corresponding ECUs. We did look into bluetooth low energy as a possible medium for data transmission, but found that it was more of a feature rather than a core function, so we decided to hold off on it until we had a completely functioning product first.

Parts List and Costs

TABLE X

ULTRABLUE TRACTOR HACKING COST ESTIMATES

Description	Quantity	Cost ($)	Comments
Raspberry Pi 4 Model B 8GB DDR4	2	$88.31	Total cost includes product cost, 7.75% sales tax, and standard shipping
Arteckin J1939 9 Pin Female to OBD-2 Cable	2	$19.65	Total cost includes product cost, 7.75% sales tax, and standard shipping
Mini HDMI to HDMI Cable 6.6ft	2	$5.38	Total cost includes product cost, 7.75% sales tax, and standard shipping
EVO Select microSDXC Memory Card 256GB	2	$32.31	Total cost includes product cost, 7.75% sales tax, and standard shipping
Pi 4 Heatsink Set of 4 - Designed for Raspberry Pi 4	2	$5.38	Total cost includes product cost and 7.75% sales tax
OBDSOL STN2100 Multiprotocol OBD to UART Interpreter IC	2	$10.76	Total cost includes product cost, 7.75% sales tax, and standard shipping
Multicolored GPIO Wires	2	$7.52	Total cost includes product cost, 7.75% sales tax, and standard shipping
2-Channel CAN-BUS (FD) Shield for Raspberry Pi	2	$30.06	Total cost includes product cost, 7.75% sales tax, and standard shipping. Not used in final product
RAK4600 Breakout Board	2	$19.40	Total cost includes product cost, 7.75% sales tax, and standard shipping. Not used in final product

Breadboards Kit	2	$8.61	Total cost includes product cost, 7.75% sales tax, and standard shipping
QFN28 to DIP28 Converter	2	$10.23	Total cost includes product cost, 7.75% sales tax, and standard shipping
Components for STN2100 Driving Circuit (Resistors, Capacitors, etc.)	2	$10.78	Total cost includes product cost, 7.75% sales tax, and standard shipping
MCP2562 High Speed CAN Transceiver	2	$1.02	Total cost includes product cost, 7.75% sales tax, and standard shipping
Kingst LA1010 Logic Analyzer	2	$62.50	Total cost includes product cost, 7.75% sales tax, and standard shipping
Total Cost		**$629.22**	

Software

Python-OBD Library Modifications

python-obd/obd/elm327.py

```
import re
import serial
import time
import logging
from .protocols import *
from .utils import OBDStatus

logger = logging.getLogger(__name__)


class ELM327:
    """
        Handles communication with the ELM327 adapter.

        After instantiation with a portname (/dev/ttyUSB0, etc...),
        the following functions become available:

            send_and_parse()
            close()
            status()
            port_name()
            protocol_name()
            ecus()
    """

    ELM_PROMPT = b'>'
    ELM_LP_ACTIVE = b'OK'

    _SUPPORTED_PROTOCOLS = {
        # "0" : None,
        # Automatic Mode. This isn't an actual protocol. If the
        # ELM reports this, then we don't have enough
        # information. see auto_protocol()
        "1": SAE_J1850_PWM,
        "2": SAE_J1850_VPW,
        "3": ISO_9141_2,
        "4": ISO_14230_4_5baud,
        "5": ISO_14230_4_fast,
        "6": ISO_15765_4_11bit_500k,
        "7": ISO_15765_4_29bit_500k,
        "8": ISO_15765_4_11bit_250k,
        "9": ISO_15765_4_29bit_250k,
        "A": SAE_J1939,
        # "B" : None, # user defined 1
        # "C" : None, # user defined 2
    }

    # used as a fallback, when ATSP0 doesn't cut it
    _TRY_PROTOCOL_ORDER = [
        "6",  # ISO_15765_4_11bit_500k
        "8",  # ISO_15765_4_11bit_250k
        "1",  # SAE_J1850_PWM
        "7",  # ISO_15765_4_29bit_500k
        "9",  # ISO_15765_4_29bit_250k
```

```
    "2",  # SAE_J1850_VPW
    "3",  # ISO_9141_2
    "4",  # ISO_14230_4_5baud
    "5",  # ISO_14230_4_fast
    "A",  # SAE_J1939
]

# 38400, 9600 are the possible boot bauds (unless reprogrammed via
# PP 0C).  19200, 38400, 57600, 115200, 230400, 500000 are listed on
# p.46 of the ELM327 datasheet.
#
# Once pyserial supports non-standard baud rates on platforms other
# than Linux, we'll add 500K to this list.
#
# We check the two default baud rates first, then go fastest to
# slowest, on the theory that anyone who's using a slow baud rate is
# going to be less picky about the time required to detect it.
_TRY_BAUDS = [38400, 9600, 230400, 115200, 57600, 19200]

def __init__(self, portname, baudrate, protocol, timeout,
             check_voltage=True, start_low_power=False):
    """Initializes port by resetting device and gettings supported PIDs. """

    logger.info("Initializing ELM327: PORT=%s BAUD=%s PROTOCOL=%s" %
                (
                    portname,
                    "auto" if baudrate is None else baudrate,
                    "auto" if protocol is None else protocol,
                ))

    self.__status = OBDStatus.NOT_CONNECTED
    self.__port = None
    self.__protocol = UnknownProtocol([])
    self.__low_power = False
    self.timeout = timeout

    # ------------- open port -------------
    try:
        self.__port = serial.serial_for_url(portname,
                                            parity=serial.PARITY_NONE,
                                            stopbits=1,
                                            bytesize=8,
                                            timeout=10)  # seconds
    except serial.SerialException as e:
        self.__error(e)
        return
    except OSError as e:
        self.__error(e)
        return

    # If we start with the IC in the low power state we need to wake it up
    if start_low_power:
        self.__write(b" ")
        time.sleep(1)

    # ----------------------- find the ELM's baud -----------------------

    if not self.set_baudrate(baudrate):
        self.__error("Failed to set baudrate")
        return

    # --------------------------- ATZ (reset) ---------------------------
    try:
```

```
        self.__send(b"ATZ", delay=1)  # wait 1 second for ELM to initialize
        # return data can be junk, so don't bother checking
    except serial.SerialException as e:
        self.__error(e)
        return

    # ------------------------- ATE0 (echo OFF) -------------------------
    r = self.__send(b"ATE0")
    if not self.__isok(r, expectEcho=True):
        self.__error("ATE0 did not return 'OK'")
        return

    r = self.__send(b"ATPP 2B 02")    # Comment
    r = self.__send(b"ATPP 2B ON")    # these
    r = self.__send(b"ATPPS")         # after
    r = self.__send(b"ATZ")           # running once
    r = self.__send(b"ATPPS")
    r = self.__send(b"STPBR 250000")
    r = self.__send(b"STPBRR")
    r = self.__send(b"STP 42")
    r = self.__send(b"STI")
    r = self.__send(b"STPO")
    r = self.__send(b"STPR")
    r = self.__send(b"STPRS")
    r = self.__send(b"ATR 0")

    # ------------------------- ATH1 (headers ON) -------------------------
    r = self.__send(b"ATH1")
    if not self.__isok(r):
        self.__error("ATH1 did not return 'OK', or echoing is still ON")
        return

    # ------------------------ ATL0 (linefeeds OFF) ------------------------
    r = self.__send(b"ATL0")
    if not self.__isok(r):
        self.__error("ATL0 did not return 'OK'")
        return


    # by now, we've successfuly communicated with the ELM, but not the car
    self.__status = OBDStatus.ELM_CONNECTED

    # ------------------------- AT RV (read volt) -------------------------
    if check_voltage:
        r = self.__send(b"AT RV")
        if not r or len(r) != 1 or r[0] == '':
            self.__error("No answer from 'AT RV'")
            return
        try:
            if float(r[0].lower().replace('v', '')) < 6:
                logger.error("OBD2 socket disconnected")
                return
        except ValueError as e:
            self.__error("Incorrect response from 'AT RV'")
            return
        # by now, we've successfuly connected to the OBD socket
        self.__status = OBDStatus.OBD_CONNECTED

    # try to communicate with the car, and load the correct protocol parser
    if self.set_protocol(protocol):
        self.__status = OBDStatus.CAR_CONNECTED
        logger.info("Connected Successfully: PORT=%s BAUD=%s PROTOCOL=%s" %
                    (
```

```
                            portname,
                            self.__port.baudrate,
                            self.__protocol.ELM_ID,
                        ))
            else:
                if self.__status == OBDStatus.OBD_CONNECTED:
                    logger.error("Adapter connected, but the ignition is off")
                else:
                    logger.error("Connected to the adapter, "
                                 "but failed to connect to the vehicle")

    def set_protocol(self, protocol_):
        if protocol_ is not None:
            # an explicit protocol was specified
            if protocol_ not in self._SUPPORTED_PROTOCOLS:
                logger.error("%s is not a valid protocol. Please use \"1\" through
\"A\"")
                return False
            return self.manual_protocol(protocol_)
        else:
            # auto detect the protocol
            return self.auto_protocol()

    def manual_protocol(self, protocol_):
        r = self.__send(b"ATTP" + protocol_.encode())
        r0100 = self.__send(b"0100")

        if not self.__has_message(r0100, "UNABLE TO CONNECT"):
            # success, found the protocol
            self.__protocol = self._SUPPORTED_PROTOCOLS[protocol_](r0100)
            return True

        return False

    def auto_protocol(self):
        """
            Attempts communication with the car.

            If no protocol is specified, then protocols at tried with `ATTP`

            Upon success, the appropriate protocol parser is loaded,
            and this function returns True
        """

        # -------------- try the ELM's auto protocol mode --------------
        r = self.__send(b"ATSP0")

        # -------------- 0100 (first command, SEARCH protocols) --------------
        r0100 = self.__send(b"0100")
        if self.__has_message(r0100, "UNABLE TO CONNECT"):
            logger.error("Failed to query protocol 0100: unable to connect")
            return False

        # ------------------- ATDPN (list protocol number) -------------------
        r = self.__send(b"ATDPN")
        if len(r) != 1:
            logger.error("Failed to retrieve current protocol")
            return False

        p = r[0]  # grab the first (and only) line returned
        # suppress any "automatic" prefix
        p = p[1:] if (len(p) > 1 and p.startswith("A")) else p
```

```
        # check if the protocol is something we know
        if p in self._SUPPORTED_PROTOCOLS:
            # jackpot, instantiate the corresponding protocol handler
            self.__protocol = self._SUPPORTED_PROTOCOLS[p](r0100)
            return True
        else:
            # an unknown protocol
            # this is likely because not all adapter/car combinations work
            # in "auto" mode. Some respond to ATDPN responded with "0"
            logger.debug("ELM responded with unknown protocol. Trying them
one-by-one")

            for p in self._TRY_PROTOCOL_ORDER:
                r = self.__send(b"ATTP" + p.encode())
                r0100 = self.__send(b"0100")
                if not self.__has_message(r0100, "UNABLE TO CONNECT"):
                    # success, found the protocol
                    self.__protocol = self._SUPPORTED_PROTOCOLS[p](r0100)
                    return True

        # if we've come this far, then we have failed...
        logger.error("Failed to determine protocol")
        return False

    def set_baudrate(self, baud):
        if baud is None:
            # when connecting to pseudo terminal, don't bother with auto baud
            if self.port_name().startswith("/dev/pts"):
                logger.debug("Detected pseudo terminal, skipping baudrate setup")
                return True
            else:
                return self.auto_baudrate()
        else:
            self.__port.baudrate = baud
            return True

    def auto_baudrate(self):
        """
        Detect the baud rate at which a connected ELM32x interface is operating.
        Returns boolean for success.
        """

        # before we change the timout, save the "normal" value
        timeout = self.__port.timeout
        self.__port.timeout = self.timeout  # we're only talking with the ELM, so
things should go quickly

        for baud in self._TRY_BAUDS:
            self.__port.baudrate = baud
            self.__port.flushInput()
            self.__port.flushOutput()

            # Send a nonsense command to get a prompt back from the scanner
            # (an empty command runs the risk of repeating a dangerous command)
            # The first character might get eaten if the interface was busy,
            # so write a second one (again so that the lone CR doesn't repeat
            # the previous command)

            # All commands should be terminated with carriage return according
            # to ELM327 and STN11XX specifications
            self.__port.write(b"\x7F\x7F\r")
            self.__port.flush()
            response = self.__port.read(1024)
```

```
            logger.debug("Response from baud %d: %s" % (baud, repr(response)))

            # watch for the prompt character
            if response.endswith(b">"):
                logger.debug("Choosing baud %d" % baud)
                self.__port.timeout = timeout  # reinstate our original timeout
                return True

        logger.debug("Failed to choose baud")
        self.__port.timeout = timeout  # reinstate our original timeout
        return False

    def __isok(self, lines, expectEcho=False):
        if not lines:
            return False
        if expectEcho:
            # don't test for the echo itself
            # allow the adapter to already have echo disabled
            return self.__has_message(lines, 'OK')
        else:
            return len(lines) == 1 and lines[0] == 'OK'

    def __has_message(self, lines, text):
        for line in lines:
            if text in line:
                return True
        return False

    def __error(self, msg):
        """ handles fatal failures, print logger.info info and closes serial """
        self.close()
        logger.error(str(msg))

    def port_name(self):
        if self.__port is not None:
            return self.__port.portstr
        else:
            return ""

    def status(self):
        return self.__status

    def ecus(self):
        return self.__protocol.ecu_map.values()

    def protocol_name(self):
        return self.__protocol.ELM_NAME

    def protocol_id(self):
        return self.__protocol.ELM_ID

    def low_power(self):
        """
            Enter Low Power mode

            This command causes the ELM327 to shut off all but essential
            services.

            The ELM327 can be woken up by a message to the RS232 bus as
            well as a few other ways. See the Power Control section in
            the ELM327 datasheet for details on other ways to wake up
            the chip.
```

```
        Returns the status from the ELM327, 'OK' means low power mode
        is going to become active.
    """

    if self.__status == OBDStatus.NOT_CONNECTED:
        logger.info("cannot enter low power when unconnected")
        return None

    lines = self.__send(b"ATLP", delay=1)

    if 'OK' in lines:
        logger.debug("Successfully entered low power mode")
        self.__low_power = True
    else:
        logger.debug("Failed to enter low power mode")

    return lines

def normal_power(self):
    """
        Exit Low Power mode

        Send a space to trigger the RS232 to wakeup.

        This will send a space even if we aren't in low power mode as
        we want to ensure that we will be able to leave low power mode.

        See the Power Control section in the ELM327 datasheet for details
        on other ways to wake up the chip.

        Returns the status from the ELM327.
    """
    if self.__status == OBDStatus.NOT_CONNECTED:
        logger.info("cannot exit low power when unconnected")
        return None

    lines = self.__send(b" ")

    # Assume we woke up
    logger.debug("Successfully exited low power mode")
    self.__low_power = False

    return lines

def close(self):
    """
        Resets the device, and sets all
        attributes to unconnected states.
    """

    self.__status = OBDStatus.NOT_CONNECTED
    self.__protocol = None

    if self.__port is not None:
        logger.info("closing port")
        self.__write(b"ATZ")
        self.__port.close()
        self.__port = None

def send_and_parse(self, cmd):
    """
        send() function used to service all OBDCommands
```

```
            Sends the given command string, and parses the
            response lines with the protocol object.

            An empty command string will re-trigger the previous command

            Returns a list of Message objects
        """

        if self.__status == OBDStatus.NOT_CONNECTED:
            logger.info("cannot send_and_parse() when unconnected")
            return None

        # Check if we are in low power
        if self.__low_power == True:
            self.normal_power()

        lines = self.__send(cmd)
        messages = self.__protocol(lines)
        return messages

    def __send(self, cmd, delay=None):
        """
            unprotected send() function

            will __write() the given string, no questions asked.
            returns result of __read() (a list of line strings)
            after an optional delay.
        """

        self.__write(cmd)

        if delay is not None:
            logger.debug("wait: %d seconds" % delay)
            time.sleep(delay)

        return self.__read()

    def __write(self, cmd):
        """
            "low-level" function to write a string to the port
        """

        if self.__port:
            cmd += b"\r"  # terminate with carriage return in accordance with ELM327
and STN11XX specifications
            logger.debug("write: " + repr(cmd))
            try:
                self.__port.flushInput()  # dump everything in the input buffer
                self.__port.write(cmd)  # turn the string into bytes and write
                self.__port.flush()  # wait for the output buffer to finish
transmitting
            except Exception:
                self.__status = OBDStatus.NOT_CONNECTED
                self.__port.close()
                self.__port = None
                logger.critical("Device disconnected while writing")
                return
        else:
            logger.info("cannot perform __write() when unconnected")

    def __read(self):
        """
            "low-level" read function
```

```
        accumulates characters until the prompt character is seen
        returns a list of [/r/n] delimited strings
    """
    if not self.__port:
        logger.info("cannot perform __read() when unconnected")
        return []

    buffer = bytearray()

    while True:
        # retrieve as much data as possible
        try:
            data = self.__port.read(self.__port.in_waiting or 1)
        except Exception:
            self.__status = OBDStatus.NOT_CONNECTED
            self.__port.close()
            self.__port = None
            logger.critical("Device disconnected while reading")
            return []

        # if nothing was received
        if not data:
            logger.warning("Failed to read port")
            break

        buffer.extend(data)

        # end on chevron (ELM prompt character) or an 'OK' which
        # indicates we are entering low power state
        if self.ELM_PROMPT in buffer or self.ELM_LP_ACTIVE in buffer:
            break

    # log, and remove the "bytearray(   ...   )" part
    logger.debug("read: " + repr(buffer)[10:-1])

    # clean out any null characters
    buffer = re.sub(b"\x00", b"", buffer)

    # remove the prompt character
    if buffer.endswith(self.ELM_PROMPT):
        buffer = buffer[:-1]

    # convert bytes into a standard string
    string = buffer.decode("utf-8", "ignore")

    # splits into lines while removing empty lines and trailing spaces
    lines = [s.strip() for s in re.split("[\r\n]", string) if bool(s)]

    return lines
```

Clear/Reset DTCs

```
import obd

obd.logger.setLevel(obd.logging.DEBUG)
connection = obd.OBD(portstr="/dev/ttyS0", baudrate="9600", protocol="A", fast=False)
#Baud rate can be set, "None" will auto select
command = obd.commands.CLEAR_DTC
```

```
response=connection.query(command)

print(response)    # Shouldn't print anything
```

Pull and Display DTCs

```
import obd

obd.logger.setLevel(obd.logging.DEBUG)
connection = obd.OBD(portstr="/dev/ttyS0", baudrate="9600", protocol="A", fast=False)
# Baud rate can be set, "None" will auto select
command = obd.commands.GET_DTC
response = connection.query(command)

if response.value is None:
    print("No DTCs")
else:
    for DTC in response.value:      # For debugging purposes
        print(DTC)
```

Live Data

```
import obd
import time

# Reference https://python-obd.readthedocs.io/en/latest/Command%20Tables/ for
parameter aliases

obd.logger.setLevel(obd.logging.DEBUG)
connection = obd.OBD(portstr="COM6", baudrate=”9600”', protocol="A", fast=False)  #
Baud rate can be set, "None" will auto select

engine_temp = obd.commands.COOLANT_TEMP;
speed = obd.commands.SPEED;
rpm = obd.commands.RPM;
fuel_level = obd.commands.FUEL_LEVEL;
oil_temp = obd.commands.OIL_TEMP;

commands = [engine_temp, speed, rpm, fuel_level, oil_temp]

while 1:
    for command in commands:
        response = connection.query(command)
        print(str(response.command) + ": " + str(response.value) + "\n")

    time.sleep(3);    # Wait 3 seconds before requesting data, can be adjusted as
needed
```

Hardware Configuration/Layout

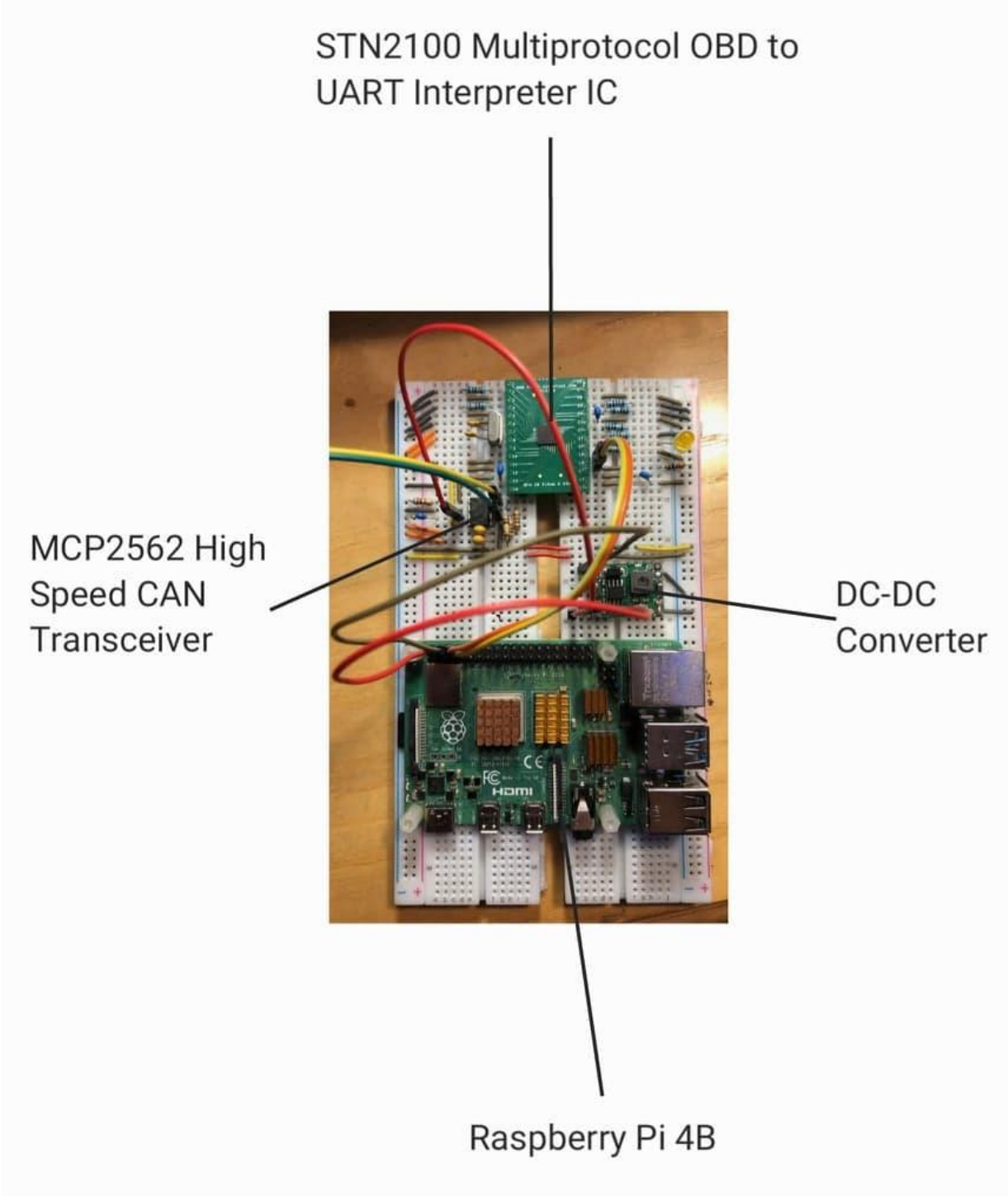


Figure 21: Hardware Configuration with Labeled Components

Figure 21 depicts the hardware layout for this project. The DC-DC converter can be any voltage regulating component that steps 5V down to 3.3V. The connections made for each component can be found in Chapter V under the Hardware section.

Customer Interviews

Louie Bayer

Louie Bayer is Cal Poly's tractor foreman. He has a lot of experience repairing and maintaining Cal Poly's tractors. We interviewed him about what an ideal diagnostic tool would be for him, and what features he would like to see in one. From this interview, we were able to generate many engineering specifications and marketing requirements. The information we gained from this interview was invaluable in starting the design of this project. We learned that as a tractor mechanic, he wanted a diagnostic tool that would present more information than just a code. He wanted the ability to clear the code, determine how often a code is reported, and just more end-user access to the tractor. He also wanted something that was able to transmit data wirelessly, whether it was via wifi or bluetooth. His ideal size of the device was the size of a portable power bank, while also being as durable as phones in that most are dust proof and water resistant. He also told us that in theory, general diagnostics are nice to have, but in practice, it is seldom used. It would be useful to him to have health checks for filters, fluids, and certain components, and to have predictive service. Not only did he provide us with great insight into the design of our project, he also gave us some important advice and background knowledge about tractors and their codes. We learned that different tractors can have different codes. We also learned that there are different conditions of codes, in that yellow meant a simple error, and that red meant that the vehicle would be non operational. We were informed that different manufacturers have different diagnostic steps and tools.

After the initial interview with Louie, we were able to put together the engineering specifications and marketing requirements and present those to him. He agreed with what we had, and was excited to see how our project progressed. Whenever he had time, we would update him on our progress and he was happy to secure tractors to perform tests on and help us out however else we needed.

Joe McKee

Joe McKee is a heavy machinery equipment salesman. As a salesman, he purchases and sells equipment that sometimes have error codes. He has experience in dealing with heavy machinery that have error codes, and has even used diagnostic tools and software provided by manufacturers. We presented to him our marketing requirements and engineering specifications and he mostly agreed with everything, with the exception of implementing LoRa.